```java
import java.util.Iterator;

public class DoubleLinkedList<ELEMENT extends Comparable <ELEMENT>> implements
ILinkedList<ELEMENT> {

  //region Node Class

  protected class Node<ELEMENT> {
    protected ELEMENT item;
    protected Node<ELEMENT> next;
    protected Node<ELEMENT> prev;

    protected Node() {
      this(null, null, null);
    }

    protected Node(ELEMENT item) {
      this(item, null, null);
    }

    protected Node(ELEMENT item, Node<ELEMENT> next) {
      this(item, next, null);
    }

    protected Node(ELEMENT item, Node<ELEMENT> next, Node<ELEMENT> prev) {
      this.item = item; // Referencia al elemento almacenado
      this.next = next; // Referencia al el siguiente elemento
      this.prev = prev; // Referencia al elemento anterior
    }

    @Override
    public String toString() {
      return this.item.toString();
    }
  }
  //endregion

  //region Attributes

  private Node<ELEMENT> head; // Referencia al primer Nodo
  private int count; // Referencia a la cantidad de Nodos
  private Node<ELEMENT> tail; // Referencia al ultimo elemento
  //endregion

  //region Constructors

  public DoubleLinkedList() {
```

```
    this.head = null;
    this.count = 0;
    this.tail = null;
}
//endregion

//region Linked List Methods

// Returns the number of elements in this list.
public int size() {
    return this.count;
}

public void addFirstRookieVersion(ELEMENT item) {
    if (this.size() <= 0) {
        this.head = this.tail = new Node<ELEMENT>(item, null, null);
        ++this.count;
    } else {
        Node<ELEMENT> temp = new Node<ELEMENT>(item, null, null);
        temp.next = this.head;
        this.head.prev = temp;
        this.head = temp;
        ++this.count;
    }
}

// Inserts the specified element at the beginning of this list.
public void addFirst(ELEMENT item) {
    Node<ELEMENT> temp = new Node<ELEMENT>(item, this.head, null);
    if (this.size() <= 0) {
        this.tail = temp;
    } else {
        this.head.prev = temp;
    }
    this.head = temp;
    ++this.count;
}

public void addLastRookieVersion(ELEMENT item) {
    if (this.size() <= 0) {
        this.head = this.tail = new Node<ELEMENT>(item, null, null);
        ++this.count;
    } else {
        Node<ELEMENT> temp = new Node<ELEMENT>(item, null, null);
        temp.prev = this.tail;
        this.tail.next = temp;
        this.tail = temp;
        ++this.count;
    }
```

```
}

// Appends the specified element to the end of this list.
public void addLast(ELEMENT item) {
   Node<ELEMENT> temp = new Node<ELEMENT>(item, null, this.tail);
   if (this.size() <= 0) {
      this.head = temp;
   } else {
      this.tail.next = temp;
   }
   this.tail = temp;
   ++this.count;
}

// Removes and returns the first element from this list.
public ELEMENT removeFirst() {
   if (this.size() <= 0) {
      throw new RuntimeException("La lista está vacía...");
   }
   ELEMENT item = this.head.item;
   this.head = this.head.next;
   if (this.head == null) {
      this.tail = null;
   } else {
      this.head.prev = null;
   }
   --this.count;
   return item;
}

// Removes and returns the last element from this list.
public ELEMENT removeLast() {
   if (this.size() <= 0) {
      throw new RuntimeException("La lista está vacía...");
   }
   ELEMENT item = this.tail.item;
   if (this.head.next == null) {
      this.head = this.tail = null;
   } else {
      this.tail = this.tail.prev;
      this.tail.next = null;
   }
   --this.count;
   return item;
}
//endregion

//region Object Methods
```

```java
  @Override
  public String toString() {

    if (this.size() <= 0) {
      return "";
    }

    // from https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/
StringBuilder.html
    StringBuilder sb = new StringBuilder();

    sb.append("[" + this.head.item.toString());
    for (Node<ELEMENT> skip = this.head.next; skip != null; skip = skip.next) {
      sb.append(", " + skip.item.toString());
    }
    sb.append("]");

    return sb.toString();
  }
  //endregion

  //region Iterable Methods
  @Override
  public Iterator<ELEMENT> iterator() {
    return new DoubleLinkedListIterator(this.head);
  }

  public class DoubleLinkedListIterator implements Iterator<ELEMENT> {
    private Node<ELEMENT> current;

    public DoubleLinkedListIterator(Node<ELEMENT> current) {
      this.current = current;
    }

    @Override
    public boolean hasNext() {
      return this.current != null;
    }

    @Override
    public ELEMENT next() {
      if (!this.hasNext()) {
        throw new RuntimeException("La lista está vacía...");
      }
      ELEMENT item = this.current.item;
      this.current = this.current.next;
      return item;
    }
```

```java
  }

  public Iterator<ELEMENT> iteratorBack() {
    return new DoubleLinkedListIteratorBack(this.tail);
  }

  public class DoubleLinkedListIteratorBack implements Iterator<ELEMENT> {
    private Node<ELEMENT> current;

    public DoubleLinkedListIteratorBack(Node<ELEMENT> current) {
      this.current = current;
    }

    @Override
    public boolean hasNext() {
      return this.current != null;
    }

    @Override
    public ELEMENT next() {
      if (!this.hasNext()) {
        throw new RuntimeException("La lista está vacía...");
      }
      ELEMENT item = this.current.item;
      this.current = this.current.prev;
      return item;
    }

  }

  // AGREGAR ELEMENTOS EN ORDEN -> Ascendente
  public void AddInOrder(ELEMENT item) { // Se recibe al elemento que se va a agregar de manera ordenada
    if (this.count == 0) { // Si la lista esta vacía
      this.head = this.tail = new Node<ELEMENT>(item, null, null); // Se pasa a ambos (head y tail) el mismo valor
      // Se referencia a los tres parámetros -> elemento a almacenar, referencia al siguiente y al anterior
      ++this.count; // Se incrementa la cantidad de nodos
    } else { // Caso contrario
      if (item.compareTo(this.head.item) <= 0) { // Si el elemento a insertar es menor (-1) / (0) o igual que head (primero)
        this.addFirst(item); // Se lo manda al principio
      } else { // Caso contrario
        if (item.compareTo(this.tail.item) > 0) { // Si el elemento es mayor (1) que tail
          addLast(item); // Se lo manda al final
        } else { // Caso contrario, se lo busca en el medio
          Node<ELEMENT> skip = this.head; // Se declara skip como puntero
```

```
            while ((skip != null && (item.compareTo(skip.item)) > 0)) {
                // Mientras skip sea válido (no se haya llegado al final de la lista)
                // y el nuevo elemento sea mayor que el actual (item -> nuevo, skip.item -> actual)
                skip = skip.next; // Entonces avanzo al siguiente nodo
                // Apunta al primer nodo cuyo valor es mayor al item que se va a agregar
            }
            if (skip == null) { // Esto no debería ocurrir
                throw new RuntimeException("Se altero el orden de los elementos");
            } else { // Caso contrario
                Node<ELEMENT> temp = new Node<ELEMENT>(item, skip, skip.prev); // Se crea el
nodo temp
                // skip apunta al siguiente nodo, y skip.prev al anterior
                skip.prev.next = temp; // Se conecta el nodo anterior con temp
                // 30.next = 50 -> 30.next = 40 -> 30 - 40 - 50
                skip.prev = temp; // Se conecta el nodo siguiente con temp
                // 50.prev = 30 -> 50.prev = 40 -> 30 - 40 - 50
                ++this.count; // Se incrementa la cantidad de nodos
            }
        }
    }
  }

  public boolean findRemove(ELEMENT item){ // True si fue encontrado y eliminado, false si no se
encontró
    if (this.size() == 0){ // Si la lista está vacía no se puede eliminar
      return false;
    }
    Node <ELEMENT> skip = this.head; // Creamos el puntero skip
    while ((skip != null) && !(item.compareTo(skip.item) == 0)){
      // Se recorre toda la lista buscando si elemento buscado es igual al que esta en la lista
      skip = skip.next; // Sigo recorriendo
    }
    if (skip == null){ // De ser null, significa que el elemento no se encontró
      return false; // Se devuelve false
    }else{ // Caso contrario
      if (skip.prev == null){ // Si el elemento no tiene nodo anterior significa que es el primero
        this.removeFirst(); // Se elimina el primero, this es una referencia al elemento actual/head
        return true; // Se confirma que se eliminó
      }else{ // Caso contrario
        if (skip.next == null){ // Si skip no tiene next, significa que esta al final
          this.removeLast(); // Se elimina el último elemento/tail
          return true; // Se confirma que se eliminó
        }else{ // De otro modo
          skip.prev.next = skip.next; // El nodo anterior ahora apunta al siguiente
          skip.next.prev = skip.prev; // El nodo siguiente ahora apunta al anterior
          skip.prev = skip.next = null; // Desconecta el nodo eliminado
          --this.count; // Decrementa la cantidad de elementos
          return true; // Se confirma que se eliminó
        }
```

```
        }
      }

    }

    // Devuelve el primer elemento de la lista, sin borrarlo

    public ELEMENT getFirst(){
      if (this.head == null){
        throw new RuntimeException("No hay elementos cargados...");
      }
      return this.head.item;
    }

    // Devuelve el último elemento de la lista, sin borrarlo

    public ELEMENT getLast(){
      if (this.head == null){
        throw new RuntimeException("No hay elementos cargados...");
      }
      return this.tail.item;
    }
    //endregion
}
```